

Processors

Computer Organization
CS 140

The Lab - Overview

The purpose of this lab is to gain understanding of processors and their behavior.

You will do this by writing code that “tickles” the processor, making it interact in a particular way; and from this you learn characteristics about that processor.

You’re doing experiments on the hardware!

In this lab, you have five very specific tasks.

1. Modify my starter code in C . Run that code on three different processors in order to determine CPI. Compare your results.
2. Compare your C and Java code on the same machine. How do they perform relative to each other?
3. Compare how your code runs in a hyper-threaded environment.
4. How does your code run on Windows vs. Linux on the same machine.
5. Use a supplied program to measure the performance penalty resulting from a control hazard in your code.

Background: Calculating CPI

In Lab 4, you measured the frequency of a PIC processor. The rules for doing that are fairly straightforward – each instruction takes 4 cycles, each branch or call takes 8 cycles. So by knowing the number of instructions and knowing the total execution time, you can calculate the cycle time.

Let's detour for a moment to talk about CPI – Cycles Per Instruction. How many cycles does it take to execute the *average* instruction? For the PIC processor, this number is between 4 and 8, depending on the number of branch instructions.

More generally, the CPI can be found by knowing the instructions executed per second and by knowing the cycles completed per second. Then:

$$\frac{\text{Cycles}}{\text{Instruction}} = \frac{(\text{Cycles} / \text{Second})}{(\text{Instructions} / \text{Second})}$$

So a “good” CPI is a small number – smaller is better. The fewer cycles required to execute an instruction, the more work gets accomplished.

Background: Useful Information

A note on compiler efficiency.

I often compile code with the ability to debug. The “-g” switch does this. So if I say “gcc -g Prog06.c -o Prog06” I get an executable where I can use gdb – and a debugger is the greatest timesaver known to the programming world.

But this doesn't produce efficient code – to get efficiency you need to compile with optimization “-O4”. While we're on the topic of switches, the “-S” switch tells the compiler to produce assembler output and NOT to produce an executable.

The cycle speed of your computer can be determined as follows:

On LINUX → the file `/proc/cpuinfo` contains the rated processor speed, as well as lots of other goodies.

On Windows Start → Control Panel → System tells you processor speed.

Appendix A.2 shows how to find the number of instructions in a cycle.

Warm-up

The code for my C and java programs is available to you – no need to type in lots of instructions. They can be found at [Prog06.c](#) and [Prog06.java](#). Here you can also find [Prog06.s](#) and [Prog06.exe](#). The source code for these programs is also listed in Appendices A and B of this document.

The Task asked of you here is to determine the processor/compiler CPI. I'm showing you how I did this task so you can see a concrete example

Look at my Prog06.c and resulting .s file. My code accomplishes a very simple task – just finding the sum of the numbers between 1 and 1,000,000.

- Try calculating the CPI of my code:
- Figure out how many instructions are required to accomplish the 1 → million sum.
- Time the number of seconds to accomplish that task.
- What is the speed of the processors – the number of cycles/second.
- Calculate the CPI
- ALL THIS IS JUST WARMUP.

Task 1: Determine the Program CPI

The Task asked of you here is to determine the processor/compiler CPI. You will measure the performance of a C (compiled) program as it runs on three different processors. The program will be accomplishing the same work on each machine, but it will require a varying amount of time.

I would VERY MUCH PREFER that you use code that you have modified – your code is accomplishing some task other than the one accomplished by My code. My code (and .exe) is available only if you get *very* stuck – you will get much more credit if you devise your own plan with your own code.

Task 1: Determine the Program CPI

1. Look at my sample C and java programs. My code accomplishes a very simple task – just finding the sum of the numbers between 1 and 1,000,000.
 - Try calculating the CPI of my code:
 - Figure out how many instructions are required to accomplish the 1 → million sum.
 - Time the number of seconds to accomplish that task.
 - What is the speed of the processors – the number of cycles/second.
 - Calculate the CPI
 - ALL THIS IS JUST WARMUP, because you should
2. Determine the CPI of a piece of your OWN code – and then plan experiments as explained later.
 - Write your **own** code to accomplish your own simple task. For example, my code does `Global += index;` Yours could do `Global *= Index;` or `Global += Index / 93;` or `Global += Index * Index;` You get the idea – there are many possibilities. Try not to pick something that your buddies are doing since there are so many choices.
 - Follow the example to determine the CPI of your own code.

Task 1: Example Calculation

Here's how I filled in the sheet when I ran a part of a Task 1 experiment.

Task 1: CPI Calculation						
Program Run	Compiler Used	Operating System				
Original Prog06.exe	gcc	Windows XP				
Processor Name	Processor Cycle Speed	Instructions per loop	Loops per reporting period	Seconds in reporting period	Instructions executed per sec.	CPI
AMD Athlon XP 3000+	2,160,000,000 / sec or 2.16 GHz	4	1,000,000,000	0.937seconds	4,268,943,436	0.506

See Appendix A2

$$4,000,000,000 / 0.937 \text{ sec} = 4,268,943,436 / \text{sec}$$

$$(2,160,000,000 \text{ cycles/sec}) / (4,268,943,436 \text{ instr. /sec})$$

Task 1: Example Calculation

Comments on the results found on the previous page:

AMD would have you believe that even though their processor runs at 2.16 GHz, it has the equivalent speed of a 3GHz – thus the marketing name. This performance may or may not be true.

Note that the CPI that I measured is about 0.5. This means that there are approximately TWO instructions executed in each processor cycle. Your numbers will vary.

Background: Experimental Design

Now you know that in any experimental design, when you have a whole list of things you can change, it's important to change only **one** variable at a time.

What are the *variables* you have in this experiment?

- a) The processor and therefore the processor cycles/second
- b) The program to be run – we've discussed three options for this so far –
 - Program.exe
 - Program – a LINUX executable (could be an apple executable also)
- c) The compiler used for your code. The compiler you run to get Program.exe might not be the one you use to get a LINUX executable.
- d) The Operating System on which you run.
- e) And there might be OTHER variables I haven't thought of here.

The POINT IS – when you design your experiment to compare three processors, you need to keep other variables the same.

Task 1: Determine the Program Performance

Reporting: In the Show and Tell section, you will find a table like this to be filled out for your three different machines.

Task 1: CPI Calculation						
Program Run	Compiler Used	Operating System				
Processor Name	Processor Cycle Speed	Instructions per loop	Loops per reporting period	Seconds in reporting period	Instructions executed per sec.	CPI

Detour: Program Portability

You will be asked to compile and run the programs you write on three different machines. You may need to do this on a windows machine. How are you going to do that? Planning is always helpful.

Let's suppose that one of the machines you want to run on is your roommate's. Let's suppose you want to run a C program. Do you have the facilities to compile a C program on your Windows box? You may need to download the gcc that runs on Windows in order to produce an .exe. Then you can easily export the .exe to your roommate's machine and execute it. This in fact makes C more portable than java in some instances.

Alternatively, you could use the Microsoft Visual Studio we have installed in the lab.

So these are all logistical details you will need to work out.

Task 2: Performance of C and Java

We want to compare the performance of C and Java. Keeping in mind our need to change only one variable, this experiment needs to be done on the same machine – you want to change ONLY language.

2) Compare C and Java Performance

Processor Name	Processor Cycle Speed	Program run	Compiler Used	Operating System	Completion Time (secs)
Intel Pentium 4 Core Duo	2.6 GHz	MyCode.c		Linux / Windows	
		MyCode.java			

The steps are simple here:

1. Modify both the C and Java program – the two codes should perform identical tasks, but those tasks must be different from my starter code.
2. Run those programs on a machine, keeping other variables as consistent as possible.
3. I've filled in some possible values in the table above.

Task 3: Determine the Program Performance

The machines we have in the lab are hyper threaded. You should read more about this, but the essence is that the processors can run two programs simultaneously (well, sort of). I would like to know what happens when you run two copies of your program at the same time – what is the execution time for each of the programs when running in this mode?

3) Run a program on a lab machine in single and hyper-threaded mode

Processor Name	Processor Cycle Speed	Program run	Compiler	Operating System	Completion Time (secs)
		C – 1 copy			
		C – 2 copies			
		Java – 1 copy			
		Java – 2 copies			

To facilitate running multiple copies of a program, you should learn about “&” and running in the *background*.

By the way, DO NOT RUN ON CSGATEWAY (younger) since this will clog up the machine. You should also learn some Useful LINUX commands including “&”, “ps aux”, “kill <pid>”, “killall Prog06”

Task 3: Determine the Program Performance

The hyper-thread experiment continued:

The hardest part of this experiment is *thinking* – sorry to be cynical, but plan this test and verify this test to make sure you have it right.

The program spits out a series of numbers – in my default case, times are printed every 1,000,000,000 cycles. DON'T use the first numbers – how do you know both programs are humming along – one of them may just be starting? Do you get the same results several times in a row – look at repeatability.

On LINUX you can do the following:

```
Prog06 &
```

```
Prog06 &
```

Which will start two programs.

You can run “top” or “vmstat 5” to see if the processors are actually being used or if something bogus is happening. Trust nothing. When you are done, type “killall Prog06” to stop the programs. They should print out the numbers and you should be fine.

Bottom line – *think*.

You will be asked to interpret these results; especially “How much more work per time is accomplished with the processes hyperthreaded rather than run singly?”

Task 4: Determine the OS Influence

So here's the question. Should the Operating System make any difference when running your code?

There is of course only one way to find out the real answer – measure it. To do this, run your C or Java program on the same machine, but change only the OS that's running. It's possible to do this with our dual-booted lab machines.

4: Measure the influence of the Operating System					
Processor Name	Processor Cycle Speed	Program run	Operating System	Compiler	Completion Time
Intel Pentium 4 Core Duo	2.6 GHz	MyCode.c OR MyCode.java		Linux etc.	
				Linux etc.	

Task 5: The cost of a control hazard.

Your task is to do the following:

Determine the number of processor cycles that are lost when a branch is mispredicted. You want one number – BUT there are a number of steps and lots of required documentation to get this number.

What you are given?

Here you work only with the program I'm providing. [Branch.c](#) can be downloaded and the source is also available in Appendix C of this document. You are also given one (1) brain.

What does the program do?

This program provides a way to outsmart the processor – to ensure that the processor can not predict whether a branch will be taken or not. This is difficult to do – the processor is VERY smart and only by going to random branching can we fool it. Look at the code in Appendix C to understand how this is done.

Task 5: The cost of a control hazard.

Step 1:

Compile the program – remember to use full optimization to produce as few instructions as possible in the loop.

Step 2:

Appendix C2 shows the code when I compiled it. Produce your own branch.s to match the way you compiled the code. Determine the number of instructions that are executed per loop in each of the three modes.

Step 3:

Measure the program execution time in the three modes.

Task 5: The cost of a control hazard.

Step 4:

Fill in the table on the next page. Bring it to lab.

Step 5:

Determine the number of cycles in an instruction loop in branch.s when the code is correctly predicted and when it is miss-predicted. How many cycles does the Pentium 4 lose on that **one** instruction as a result of miss-prediction?

Step 6:

When all else fails, and you don't understand these instructions, don't blindly follow your misunderstanding. Go back to the previous pages and **understand** the goal. Then do it the best you can.

Task 5: The cost of a control hazard.

This table can help you with the calculation

Branch Misprediction Work Sheet							
Mode	Hz	Execution Time	Instr / Loops	Instr loop	Instr / second	Machine Cycles / Loop	CPI

Mode – Which mode (0,1,2) we're running in.

Hz -- What is the cycle speed of the processor (normally in GHz)

Execution Time – How long does it take to execute the program

Instr / Loops - From the Appendix C2, how many instructions are there in a loop?

Instr loop - How many instruction loops were executed in the reported time.

Instr / second – Calculate the number of instructions executed in one second.

Machine Cycles/Loop – How many machine cycles were required to execute the number of instructions in one loop?

CPI – what is the CPI for the program running in this mode?

Evaluation Sheet Overview

Lab 06:

Timetable

Here are the tasks and when they need to happen.

Check Point	Task 1	Task 2	Task 3	Task 4	Task 5
Checkpoint 1	Do an in-class quiz showing you are familiar with the content of this lab. Happens soon after the lab kickoff.				
Checkpoint 2	Code Written – can describe the experiment to be run.	Code Written			Able to describe the *.s code.
Checkpoint 3	Experiments completed – able to answer questions as shown in the next few pages of evaluation sheets.				

Evaluation Sheet

Lab 06:

Your Name: _____

Checkpoint 1:

Result of an in-class Quiz given in the class period after Lab06 Kickoff.

1.

Evaluation Sheet

Lab 06:

Your Name: _____

Checkpoint 2:

One week after lab is assigned.

2. Task 1: Student C code is created. It compiles and runs as seen in Show & Tell.
3. Task 2: Student java code is created. It interprets and runs as seen in Show & Tell.
4. Task 1: Experiments are designed. Student can articulate the environments and conditions where the three experiments will be run.
5. Task 5: Student has compiled branch.c in order to produce branch.s. The student can explain what's happening in the branch.s code he/she brings to S & T. Which is the conditional branch that's mispredicted?

Evaluation Sheet

Lab 06:

Your Name: _____

Checkpoint 3 Task 1:

6. Student has completed and brought to lab this result matrix and corresponding worksheets showing comparison of three processors.
7. Student can explain what is happening here.

Task 1: CPI Calculation						
Program Run	Compiler Used	Operating System				
Processor Name	Processor Cycle Speed	Instructions per loop	Loops per reporting period	Seconds in reporting period	Instructions executed per sec.	CPI

Evaluation Sheet

Lab 06:

Your Name: _____

Checkpoint 3 Task 2:

8. Student has completed and brought to lab this result matrix and corresponding worksheets showing comparison of Java and C programs.
9. Student can explain the results – why do they make sense.

2) Compare C and Java Performance					
Processor Name	Processor Cycle Speed	Program run	Compiler Used	Operating System	Completion Time (secs)
		C			
		JAVA			

See example earlier for how this is filled in.

Evaluation Sheet

Lab 06:

Your Name: _____

Checkpoint 3 Task 3:

10. Student has completed and brought to lab this result matrix and corresponding worksheets showing comparison of Java and C programs.
11. Student can explain the results – why do they make sense.

3) Run a program on a lab machine in single and hyper-threaded mode					
Processor Name	Processor Cycle Speed	Program run	Compiler	Operating System	Completion Time (secs)
		C – 1 copy			
		C – 2 copies			
		Java – 1 copy			
		Java – 2 copies			

See example earlier for how this is filled in.

Evaluation Sheet

Lab 06:

Your Name: _____

Checkpoint 3 Task 4:

12. Student has completed and brought to lab this result matrix and corresponding worksheets showing comparison of LINUX and Windows programs.
13. Student can explain the results – why do they make sense.

4: Measure the influence of the Operating System

Processor Name	Processor Cycle Speed	Program run	Operating System	Compiler	Completion Time
				Linux	
				Windows	

See example earlier for how this is filled in.

Evaluation Sheet

Lab 06:

Your Name: _____

Checkpoint 3: Task 5

14. Student has correctly completed and brought to lab this result matrix and corresponding worksheet showing what happens when the branch program is run in the various modes.
15. The student can explain what she/he did.
16. The student knows the cost of mispredicting that branch instruction. How much does that ONE instruction cost when it is mispredicted?

Branch Misprediction Work Sheet

Mode	Hz	Execution Time	Instr / Loops	Instr loop	Instr / second	Machine Cycles / Loop	CPI

Appendix A.1

Prog06.c

```
/******  
This program is designed to show you how to calculate CPI. You  
will write a similar program in order to make your own measurements.  
The program should be able to run in two modes:  
1) "Prog06" - The program will run forever  
2) "Prog06 <Iterations> " - The program will do this many MILLION  
loops and then quit.  
*****/  
#include <stdio.h>  
#include <stdlib.h>  
#include <time.h>  
#include <sys/time.h>  
  
int global = 0; // This variable has scope outside of main  
  
double GetCurrentTime( ); // This is a prototype  
  
int main( int argc, char *argv[] )  
{  
    int RequestedIterations; // How many times we want to do  
                                // the operation.  
    int index; // Used to count 1,000,000 loops  
    int CompletedIterations; // Number of mega-loops completed  
  
    double StartTime, EndTime;  
  
    // If the user asks to set iterations, do it here.  
    RequestedIterations = -1;  
    if (argc > 1 )  
        RequestedIterations = atoi( argv[1] );  
  
    // Initialize time and counter  
    StartTime = GetCurrentTime( );  
    CompletedIterations = 0;
```

Appendix A.1

Prog06.c

```
while( 1 ) {
    // Do a million iterations - this is the minimum number possible
    for ( index = 0; index < 1000000; index++ ) {
        global += index;
    }
    CompletedIterations++;
    // Every once in a while, print out current time/count
    if ( ( CompletedIterations % 1000 ) == 0 ) {
        EndTime = GetCurrentTime( );
        printf( "Elapsed seconds = %11.6f: MegaLoops = %6d\n",
                EndTime - StartTime, CompletedIterations );
    }
    // If we got the requested number, then quit
    if ( ( RequestedIterations != -1 )
        && ( CompletedIterations >= RequestedIterations ) )
        break;
    } // End of while TRUE loop

    // Do a wrapup report
    printf( "Value of global = %d\n", global );
    EndTime = GetCurrentTime( );
    printf( "Elapsed seconds = %11.6f: MegaLoops = %6d\n",
            EndTime - StartTime, CompletedIterations );
    exit(0);
} // End of main */

/*****
 * Get the current time of day.
 * Time is returned in seconds and fractions of a second.
 *****/
double GetCurrentTime( ) {
    struct timeval tp;
    double TimeReturned;
    gettimeofday ( &tp, NULL );
    TimeReturned = tp.tv_usec;
    TimeReturned += tp.tv_sec*1000000;
    TimeReturned /= 1E6;
    return( TimeReturned );
} // End of GetCurrentTime */
```

**These few lines
are where all the
action takes
place.**

Appendix A.2

Prog06.s

The command `gcc -O4 -S Prog06.c` produces a file `Prog06.s`. This file contains the C code converted into Intel assembler. The lines you see below are from this file and represent the code:

```
for ( index = 0; index < 1000000; index++ ) {  
    global += index;  
}
```

```
.L21:  
    movl    global, %edx    ; Register %edx will hold the value of global.  
    xorl    %eax, %eax     ; Register %eax is index – here it's set to 0  
.L8:  
    addl    %eax, %edx     ; This is global += index;  
    addl    $1, %eax       ; This line is index++  
    cmpl    $1000000, %eax ; Compare literal - index < 1000000 – sets Z if they are the same.  
    jne     .L8            ; Go to .L8 if %eax is not equal to 1000000 – checks the Z status bit
```

Be able to explain why there are 4 instructions in the loop for this code.

Appendix B

Prog06.java

```
/******  
This program is designed to show you how to calculate CPI. You  
will write a similar program in order to make your own measurements.  
The program should be able to run in two modes:  
1) "java Prog06" - The program will run forever  
2) "java Prog06 <Iterations> " - The program will do this many MILLION  
loops and then quit.  
*****/  
public class Prog06 {  
    static int global = 0;  
  
    public static void main(String[] args) {  
        int RequestedIterations; // How many times we want to do  
                                // the operation.  
        int index; // Used to count 1,000,000 loops  
        int CompletedIterations; // Number of mega-loops completed  
        double StartTime, EndTime;  
  
        RequestedIterations = -1;  
        // If the user gives us iterations, get them here  
        if (args.length > 0) {  
            try {  
                RequestedIterations = Integer.parseInt(args[0]);  
            } catch (NumberFormatException e) {  
                System.err.println("Argument must be an integer");  
                System.exit(1);  
            }  
        }  
    }  
}
```

Appendix B

Prog06.java

```
// Initialize numbers
StartTime = (double)System.currentTimeMillis()/1000.0;
CompletedIterations = 0;
// loop here until iterations complete - this is checked later on
while( true ) {
    for ( index = 0; index < 1000000; index++ ) {
        global += index;
    }
    CompletedIterations++;
    // Every once in a while print out time/count
    if ( ( CompletedIterations % 1000 ) == 0 ) {
        EndTime = (double)System.currentTimeMillis()/1000.0;
        System.out.printf("Elapsed seconds = %11.6f  Megaloops = %6d\n",
            EndTime - StartTime,  CompletedIterations);
    }
    // Check if we're done
    if ( ( RequestedIterations != -1 )
        && ( CompletedIterations >= RequestedIterations ) )
        break;
} // End of while TRUE loop

// Do wrapup count
System.out.printf( "Value of global = %d\n", global );
EndTime = (double)System.currentTimeMillis()/1000.0;
System.out.printf("Elapsed seconds = %11.6f  Megaloops = %6d\n",
    EndTime - StartTime,  CompletedIterations);
}
/* End of main */
}
```

**These few lines
are where all the
action takes
place.**

Appendix C1

Branch.c

```
/*
*****
This program is designed to show branch prediction behavior
*****
*/

#include      <stdio.h>
#include      <stdlib.h>
#include      <sys/time.h>
#include      <time.h>

#define          NUM_DIR          4096

unsigned long   global = 0;

double   GetCurrentTime();

int   main( int argc, char *argv[] )
{
    long           branch_type;
    unsigned long  iterations; /* How many times we want to do
                               the operation.          */

    long           index, iters;
    unsigned char  direction[NUM_DIR];

    double         StartTime, EndTime;

    if (argc < 3 ) {
        printf( "Usage branch <Branch_type> <Iterations>\n" );
        printf( "Branch_type = 0 means  branch NOT taken\n" );
        printf( "Branch_type = 1 means  branch IS  taken\n" );
        printf( "Branch_type = 2 means  branch IS  random\n" );
        printf( "The total number of loops will be" );
        printf( "   %d X Iterations\n", NUM_DIR );
        exit(-1);
    }

    branch_type      = atol( argv[1] );
    iterations       = atol( argv[2] );
    printf( "Inputs: Branch Type: %d  Iterations: %d\n",
           branch_type, iterations );
}
```

Appendix C1

Branch.c

**These few lines
are where all the
action takes
place.**

```
/* Set up the direction array to determine the branches */
for ( index = 0; index < NUM_DIR; index++ )
{
    if ( branch_type == 0 )
        direction[index] = 0;
    if ( branch_type == 1 )
        direction[index] = 1;
    if ( branch_type == 2 )
        direction[index] = rand() % 2;
}
StartTime = GetCurrentTime();
for ( iters = 0; iters < iterations; iters++ )
{
    for ( index = 0; index < NUM_DIR; index++ )
    {
        if ( direction[index] == 0 )
            global = global + 3;
        else
            global = global + 2;
    }
}
/* End of for index */
/* End of for iters */
EndTime = GetCurrentTime();
printf( "Elapsed seconds = %f: Loops = %d\n",
        EndTime - StartTime, NUM_DIR * iterations );
exit(0);
/* End of main */

/*****
 * Get the current time of day.
 * Time is returned in seconds and fractions of a second.
 *****/
/
double    GetCurrentTime( )    {
    struct timeval tp;
    double TimeReturned;
    gettimeofday (&tp, NULL);
    TimeReturned = tp.tv_usec;
    TimeReturned += tp.tv_sec*1000000;
    TimeReturned /= 1E6;
    return( TimeReturned );
}
/* End of GetCurrentTime */
```

Appendix C2

Branch.s

The command `gcc -O3 -S Branch.c` produces a file `Branch.s`. This file contains the C code converted into Intel assembler. The lines you see below are from this file and represent the code:

```
for ( index = 0; index < NUM_DIR; index++ ){
    if ( direction[index] == 0 )
        global = global + 3;
    else
        global = global + 2;
}
```

```
.L17:
    xorl    %eax, %eax    ; %eax holds value for index
    jmp     .L18

.L33:
    addl    $3, %edx      ; Add 3 to global

.L21:
    addl    $1, %eax      ; This is index++
    cmpl    $4096, %eax   ; This is index < NUM_DIR (4096)
    je     .L32           ; It's equal - get out of loop

.L18:
    cmpb    $0, (%eax,%ebx) ; Is direction[] == 0?
    je     .L33           ; Yes - to L33 and add 3 to global
    addl    $2, %edx      ; No - add 2 to global
    jmp     .L21          ; Go around the loop again

.L32:
    Code continues here after loop
```